Homework 02

In this homework we work on images, colormaps and sensor calibration.

```julia
begin
    using ColorSchemes  # for different colors maps
    using Plots         # for heatmap
    using TestImages    # for testimage
    using ImageShow     # for better rendering of inline images in Pluto
    using Colors # for Gray
    using PlutoUI
end
```

`my_show`

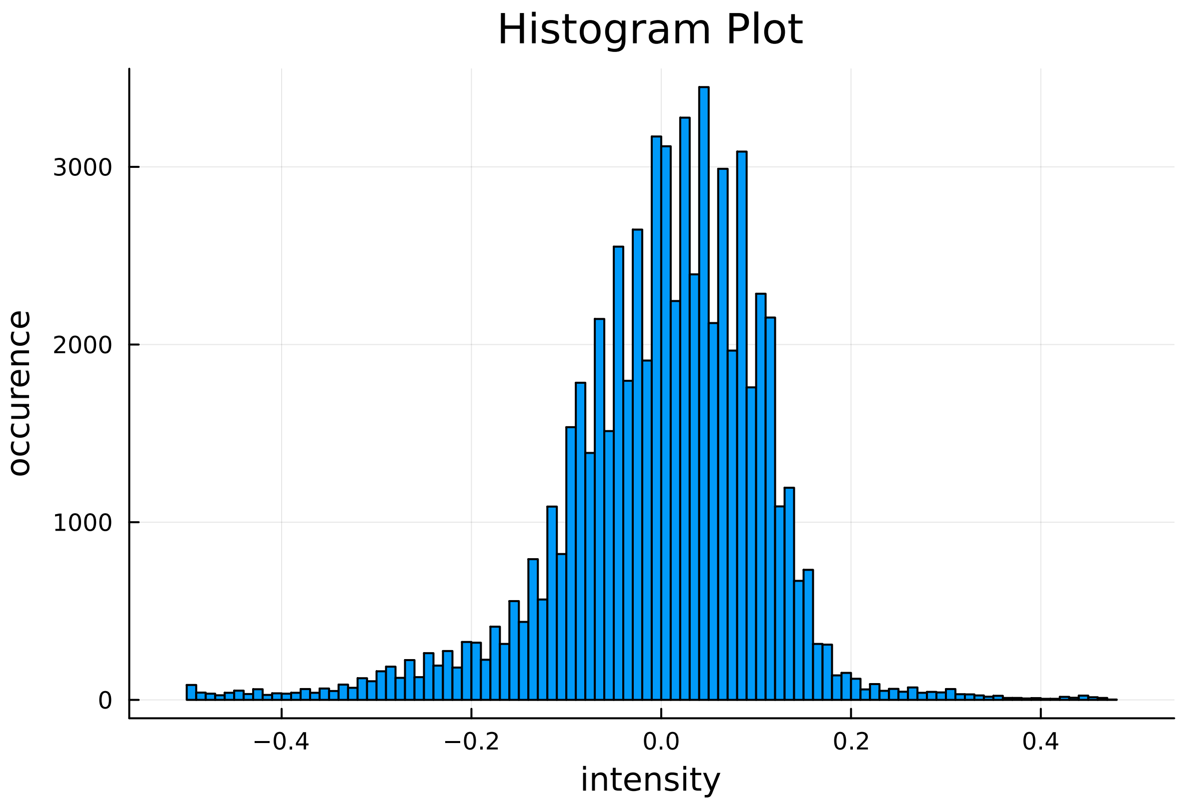
```
my_show(arr::AbstractArray{<:Real}; set_one=false, set_zero=false)
```

Displays a real valued array . Brightness encodes magnitude. Works within Jupyter and Pluto.

Keyword args

- `set_one=false` divides by the maximum to set maximum to 1
- `set_zero=false` subtracts the minimum to set minimum to 1

```julia
"""
    my_show(arr::AbstractArray{<:Real}; set_one=false, set_zero=false)
Displays a real valued array . Brightness encodes magnitude.
Works within Jupyter and Pluto.
## Keyword args
* `set_one=false` divides by the maximum to set maximum to 1
* `set_zero=false` subtracts the minimum to set minimum to 1
"""
function my_show(arr::AbstractArray{<:Real}; set_one=true, set_zero=false)
    arr = set_zero ? arr .- minimum(arr) : arr
    arr = set_one ? arr ./ maximum(arr) : arr
    Gray.(arr)
end
```

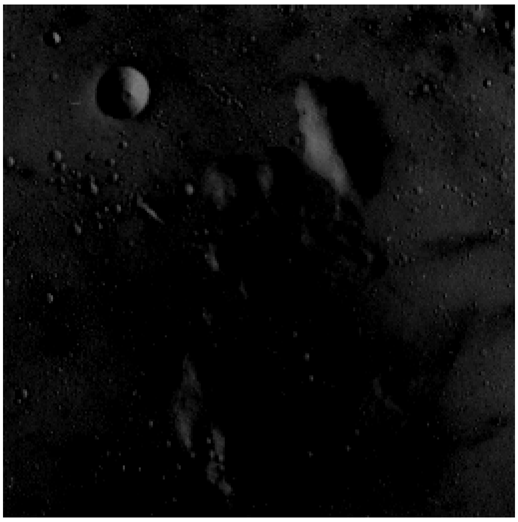



```julia
begin
    # load a image and shift it to negative values as well
    img = Float64.(testimage("moonsurface")) .- 0.5;
    histogram(img[:], xlabel="intensity", ylabel="occurence", title="Histogram
    Plot", legend=nothing)
end
```

1. Colormaps

Very often in microscopy, we look at grayscale images. We already learnt that we can use `Gray.(img)` for that. However, that is only a very simple viewer. In general, there are much more sophisticated ones like `View5D.jl` or `Napari.jl`.

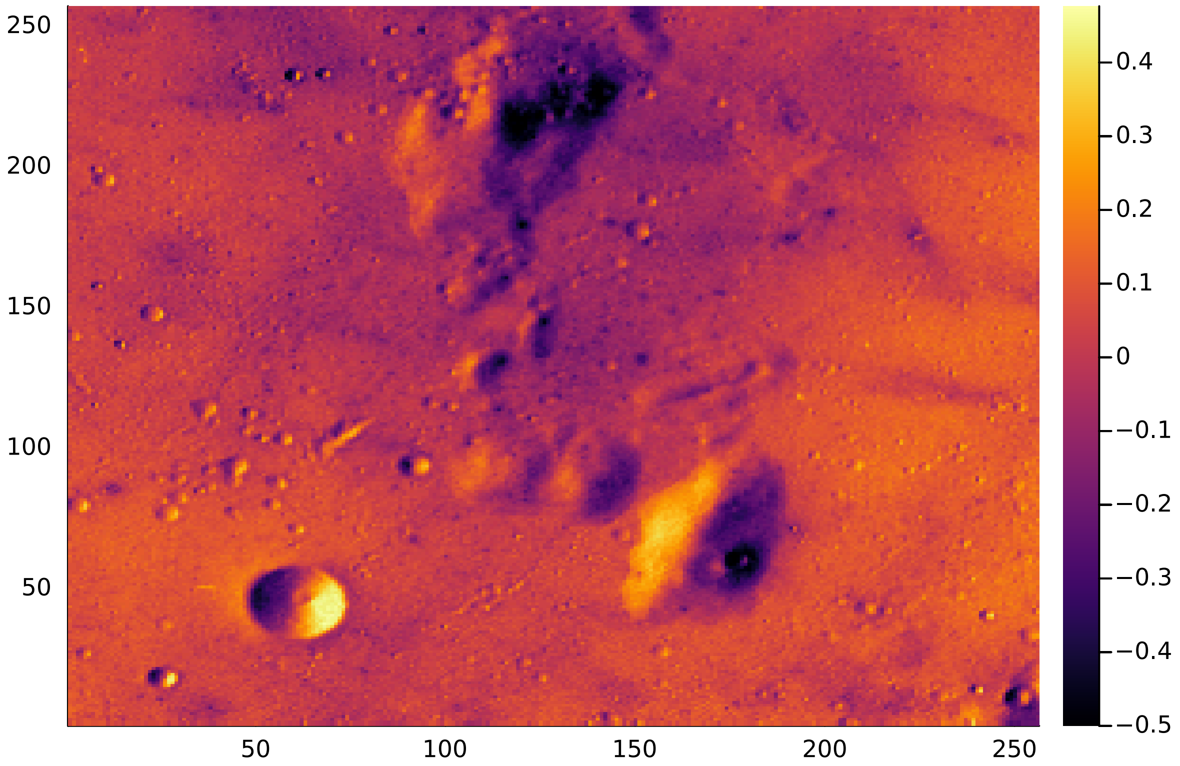
As you can see below, `Gray` fails pretty quickly and also does not make use of our human color vision.



```julia
Gray.(img)
```

1.1 Heatmap

`img` contains also negative values which `Gray` does not support by default. A nicer tool is heatmap.




```
heatmap(img)
```

1.2 Task - Equal Intensity Color Map

We now want to create a RGB color map which always has the same intensity.

Hence, `green_value + red_value + blue_value = const` for all different inputs.

to_equal_intensity_tuple

```
to_equal_intensity_tuple(value::T) where T
```

Returns for a given scalar input a tuple of equal intensity. There is no unique solution, we only require that the sum is always 1. For example, below would be a valid solution.

Examples

```
julia> to_equal_intensity_tuple(1.0)
(0.04465819873852073, 0.33333333333333326, 0.6220084679281465)

julia> to_equal_intensity_tuple(0.0)
(0.044658198738520505, 0.3333333333333333, 0.6220084679281462)

julia> to_equal_intensity_tuple(0.3f0)
(0.26402956f0, 0.6503522f0, 0.08561832f0)
```

```julia
"""
    to_equal_intensity_tuple(value::T) where T

Returns for a given scalar input a tuple of equal intensity.
There is no unique solution, we only require that the sum is always 1.
For example, below would be a valid solution.

## Examples
```julia-repl
julia> to_equal_intensity_tuple(1.0)
(0.04465819873852073, 0.33333333333333326, 0.6220084679281465)

julia> to_equal_intensity_tuple(0.0)
(0.044658198738520505, 0.3333333333333333, 0.6220084679281462)

julia> to_equal_intensity_tuple(0.3f0)
(0.26402956f0, 0.6503522f0, 0.08561832f0)
```
"""
function to_equal_intensity_tuple(x::T) where T
    # with the function signature we can access T (which is the type of `value`)
    # 0 has type Int
    # 1.0 has type Float64
    # 1.0f0 has type Float32
    if x < 0 || x > 1
        error("value must be within [0,1]")
    end

    # fix those three functions such that the property is achieved
    # maybe think about periodic functions ;)
    red_map(x) = (T(1) .+ cos(T(2π) .* x)) / T(3)

    green_map(x) = (T(1) .+ cos(T(2π) .* x + T(2π/3))) / T(3)

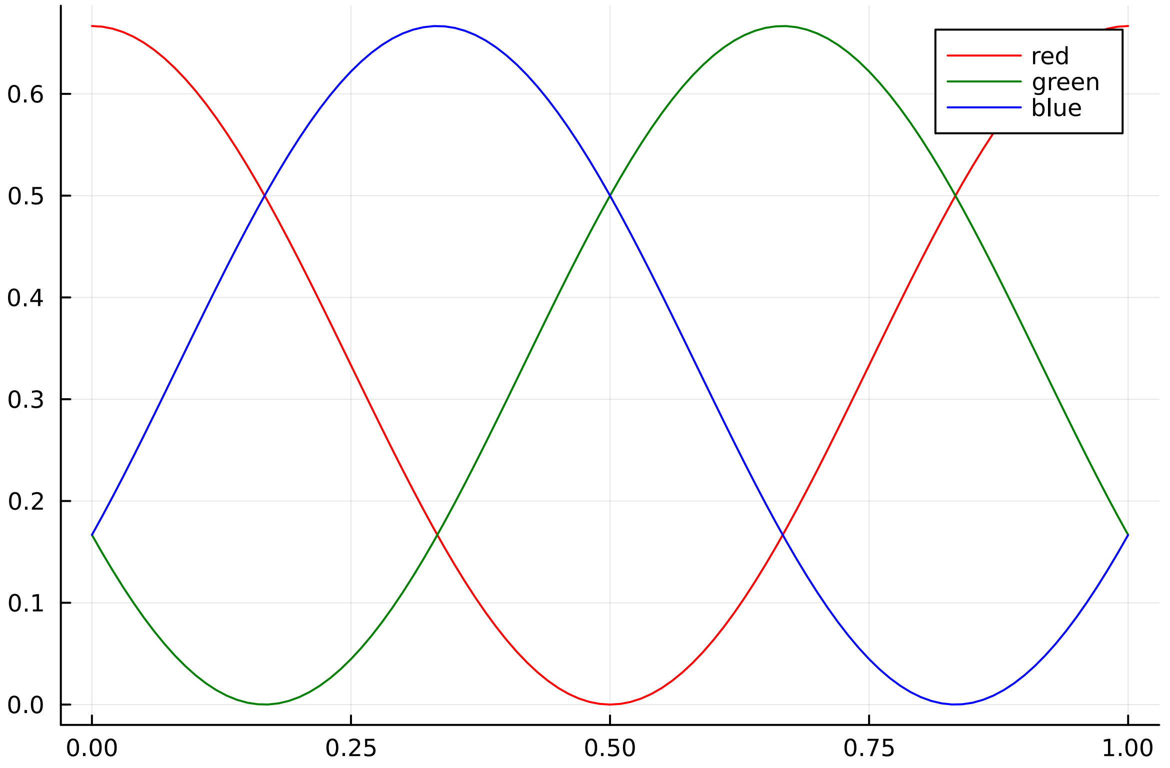
    blue_map(x) = (T(1) .+ cos(T(2π) .* x + T(4π/3))) / T(3)

    return (red_map(x), green_map(x), blue_map(x)) # TODO, return the correct tuple!
end
```

Float32

```julia
typeof(Float32.(2π/3))
```

Maybe a visualization helps




```julia
begin
    unzip(a) = map(x->getfield.(a, x), fieldnames(eltype(a)))
    x = 0:0.01:1
    r,g,b = unzip(to_equal_intensity_tuple.(x))
    plot(x, r, color=:red, label="red")
    plot!(x, g, color=:green, label="green")
    plot!(x, b, color=:blue, label="blue")
end
```

1.2 Test

```julia
using PlutoTest
```

all(to_equal_intensity_tuple(0.0f0) .≈ to_equal_intensity_tuple(1.0f0))

```julia
# cyclic
PlutoTest.@test all(to_equal_intensity_tuple(0f0) .≈ to_equal_intensity_tuple(1f0))
```

to_equal_intensity_tuple(0.1) != to_equal_intensity_tuple(0.2)

```julia
PlutoTest.@test to_equal_intensity_tuple(0.1) != to_equal_intensity_tuple(0.2)
```

1 ≈ sum(to_equal_intensity_tuple(rand(0.0f0:1.0f0)))

```julia
# intensity is 1
PlutoTest.@test 1 ≈ sum(to_equal_intensity_tuple(rand(0f0:1f0)))
```

```
all(1 .≈ sum.(to_equal_intensity_tuple.(rand(0.0:1.0, 100))))
```

```julia
# test for many values
PlutoTest.@test all(1 .≈ sum.(to_equal_intensity_tuple.(rand(0.0:1.0, 100))))
```

Check if the output is always 0<=output<=1

Side Note: it took quite a while to engineer this line :D

```
bbb = true
```

```julia
bbb = foldl(&, foldl.(Ref((acc, x) -> acc || 0<=x<=1), to_equal_intensity_tuple.
(rand(0.0:1.0, 100)), init=false))
```

```
bbb
```

```julia
PlutoTest.@test bbb
```

Type Stability

The test might be wrong. Check what happens with expresions like `2π / 3`. Which type do they have? How can we convert them to different types?

```
NTuple{3, Float32} == typeof(to_equal_intensity_tuple(rand(Float32.(0.0:1.0))))
```

```julia
PlutoTest.@test NTuple{3, Float32} == typeof(to_equal_intensity_tuple(rand(Float32.
(0.0:1.0))))
```

```
NTuple{3, Float64} == typeof(to_equal_intensity_tuple(rand(Float64.(0.0:1.0))))
```

```julia
PlutoTest.@test NTuple{3, Float64} == typeof(to_equal_intensity_tuple(rand(Float64.
(0.0:1.0))))
```

1.3 Register Color Map

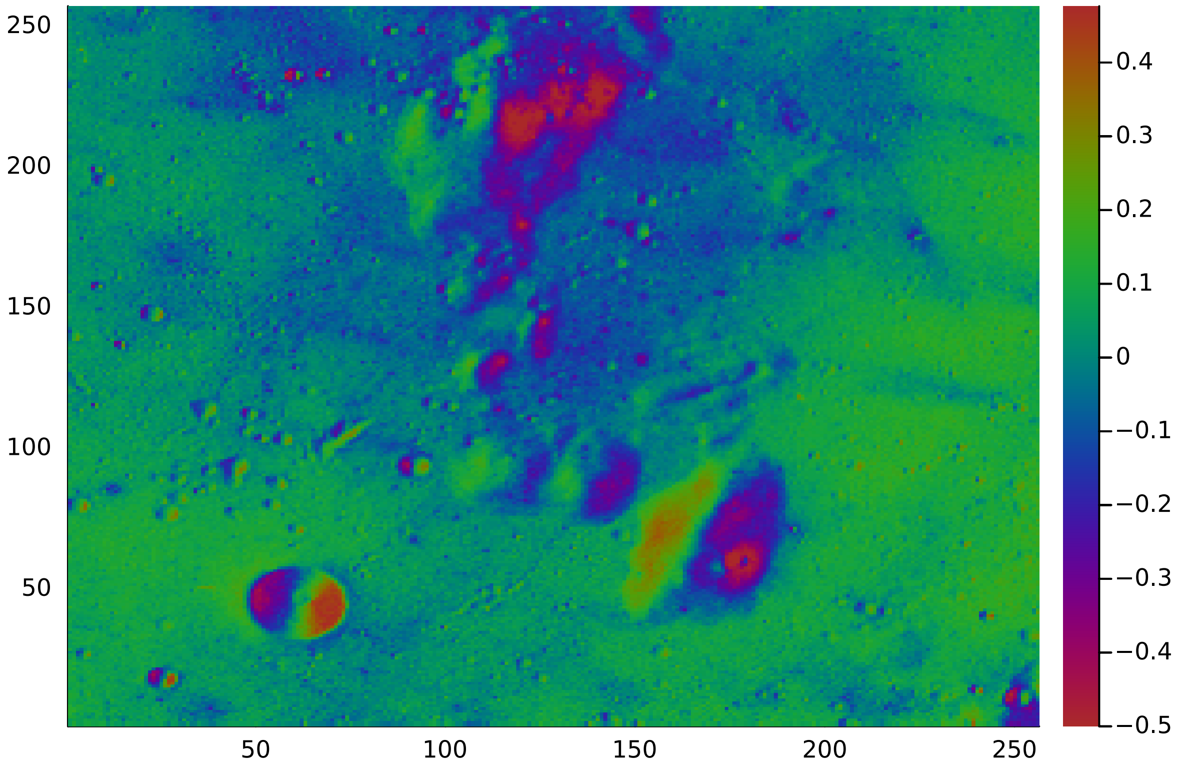
We are able to create valid outputs, but we still need to pass that to the colormap mechanism. We need to register the colormap so that we can load it with `heatmap`.

You don't need to know the details, but if you'd like understand, you find those in ColorSchemes.jl.

```
create_equal_intensity_colormap (generic function with 1 method)
```

```julia
function create_equal_intensity_colormap()
    values = range(0, 1, length=128)
    return map(x -> RGB(x...), to_equal_intensity_tuple.(values))
end
```

```julia
begin
    my_equal_intensity_colormap = create_equal_intensity_colormap()
    loadcolorscheme(:my_equal_intensity_colormap, my_equal_intensity_colormap)
end
```




```
heatmap(img, c=:my_equal_intensity_colormap)
```

1.4 Task

Now we want to create a colormap, which maps values between 0 and 0.5 to blue values and all values above 0.5 to red. Don't let you confuse by negativ and positive intensities. `loadcolorschemes` needs the values within the interval [0, 1] and will automatically scale it to the image.

`to_negative_positive_tuple`

```
to_negativ_positive_tuple(value::T) where T
```

Examples

```
julia> to_negative_positive_tuple(0.0)
(0.0, 0.0, 1.0)

julia> to_negative_positive_tuple(0.25)
(0.5, 0.5, 1.0)

julia> to_negative_positive_tuple(0.5)
(1.0, 1.0, 1.0)

julia> to_negative_positive_tuple(0.75)
(1.0, 0.5, 0.5)

julia> to_negative_positive_tuple(1.0)
(1.0, 0.0, 0.0)
```

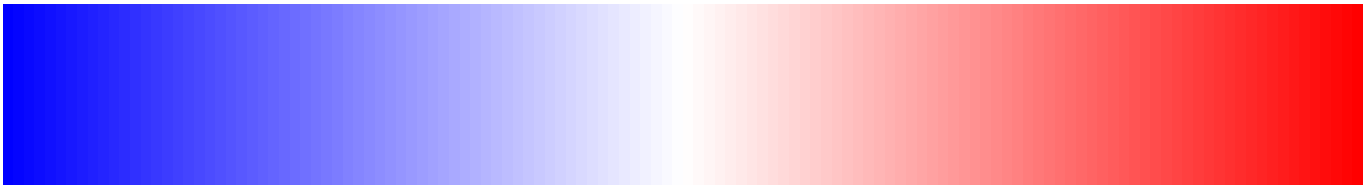
````julia
"""
    to_negativ_positive_tuple(value::T) where T


## Examples
```
julia> to_negative_positive_tuple(0.0)
(0.0, 0.0, 1.0)

julia> to_negative_positive_tuple(0.25)
(0.5, 0.5, 1.0)

julia> to_negative_positive_tuple(0.5)
(1.0, 1.0, 1.0)

julia> to_negative_positive_tuple(0.75)
(1.0, 0.5, 0.5)

julia> to_negative_positive_tuple(1.0)
(1.0, 0.0, 0.0)
```
"""
function to_negative_positive_tuple(x::T) where T

    if x < 0 || x > 1
        error("value must be within [0,1]")
    end

    if x > 0.5
        x_r = T(1)
        x_g = T(2) * (T(1) - x)
        x_b = T(2) * (T(1) - x)
        return (x_r, x_g, x_b)
    end

    if x <= 0.5
        x_r = T(2) * x
````

```
        x_g = T(2) * x
        x_b = T(1)
        return (x_r, x_g, x_b)
    end
end
```

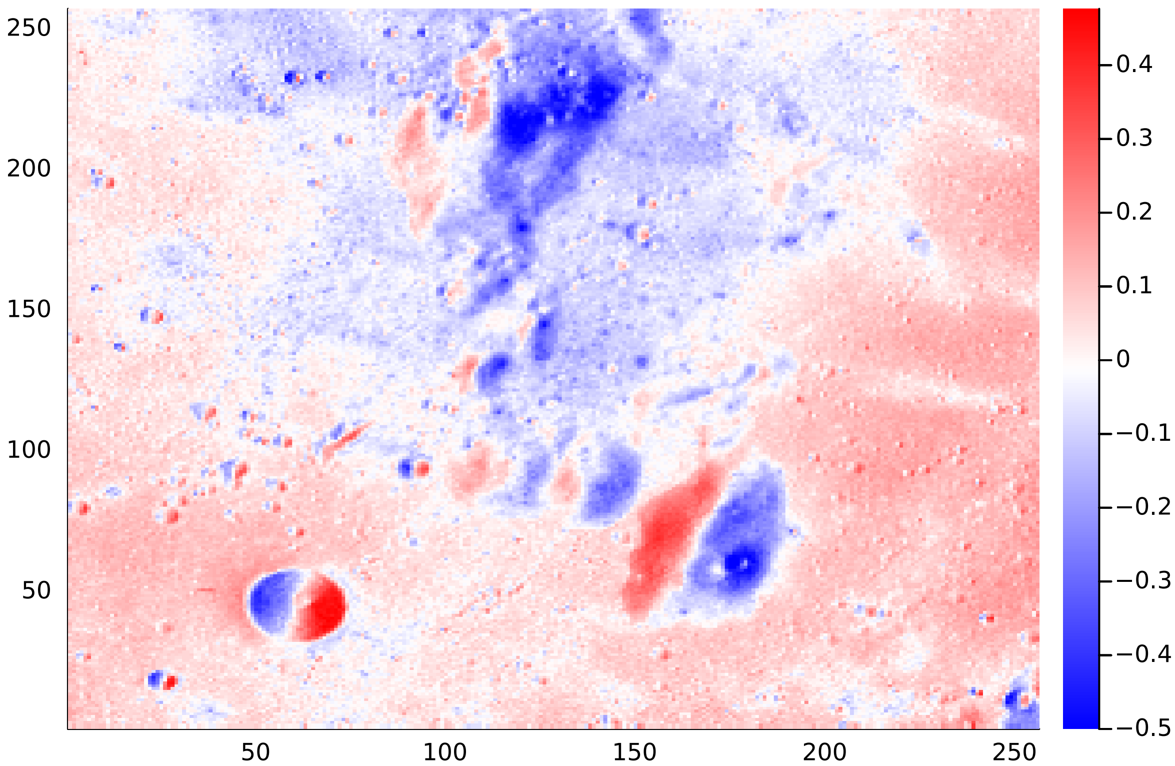
Try to show again a heatmap image. The procedure is exactly the same above, except that you need to exchange the core part of the color generation.

```
create_negative_positive_colormap (generic function with 1 method)
```

```
function create_negative_positive_colormap()
    values = range(0, 1, length=128)
    return map(x -> RGB(x...), to_negative_positive_tuple.(values))
end
```



```
begin
    my_negative_positive_colormap = create_negative_positive_colormap()
    loadcolorscheme(:my_negative_positive_colormap, my_negative_positive_colormap)
end
```




```
heatmap(img, c=:my_negative_positive_colormap)
```

1.4 Test

```
to_negative_positive_tuple(0.0) == (0.0, 0.0, 1)
```

```
PlutoTest.@test to_negative_positive_tuple(0.0) == (0.0, 0.0, 1)
```

to_negative_positive_tuple(1.0) == (1.0, 0.0, 0.0)

```
PlutoTest.@test to_negative_positive_tuple(1.0) == (1.0, 0.0, 0.0)
```

to_negative_positive_tuple(0.5) == (1.0, 1.0, 1.0)

```
PlutoTest.@test to_negative_positive_tuple(0.5) == (1.0, 1.0, 1.0)
```

to_negative_positive_tuple(0.75) == (1.0, 0.5, 0.5)

```
PlutoTest.@test to_negative_positive_tuple(0.75) == (1.0, 0.5, 0.5)
```

to_negative_positive_tuple(0.25) == (0.5, 0.5, 1)

```
PlutoTest.@test to_negative_positive_tuple(0.25) == (0.5, 0.5, 1)
```

typeof(to_negative_positive_tuple(0.25f0)) == Tuple{Float32, Float32, Float32}

```
PlutoTest.@test typeof(to_negative_positive_tuple(0.25f0)) == Tuple{Float32,
Float32, Float32}
```

typeof(to_negative_positive_tuple(0.0)) == Tuple{Float64, Float64, Float64}

```
PlutoTest.@test typeof(to_negative_positive_tuple(0.0)) == Tuple{Float64, Float64,
Float64}
```

2 Sensor Simulation

In this part, we want to simulate a sensor. Later, we try to calibrate our artifical sensor.

In short: Each pixel in the sensor measures a number of photons (`n_photons`) which is affected by Poisson noise. Afterwards, the measured value is altered by additive Gauss noise with a certain standard deviation σ. This value is then converted to ADU units (analog to digital units) with a certain `gain` (linear conversion factor). Additionally, each sensor has an `offset` which is finally added.

```
begin
    using PoissonRandom # for poisson noise ('pois_rand')
    using Statistics # for mean, var
end
```

2.1 Task

Fill in the missing parts.

simulate_pixel

```
simulate_pixel(n_photons; read_σ=5, offset=0.5, gain=0.1)
```

In this function we simulate a single pixel. It transforms a photon number to a digital unit (ADU).

The input is a integer number of photons which is then altered with poisson noise (use `pois_rand` for that). This number is changed with additive Gauss noise (see HW01). Finally, the result of this operation is multiplied by the linear `gain`. At the end, we add `offset`.

- `n_photons`: number of photons
- `read_σ`: additive read noise of the sensor (applied before `gain`)
- `gain`: how much digital output per photon
- `offset`: how much digital offset

Example Outputs - can vary in your case!

```
julia> simulate_pixel(10)
0.8570431305838304

julia> simulate_pixel(10)
2.060753446522642

julia> simulate_pixel(10)
1.020461460534897

julia> simulate_pixel(100)
8.96276245560446

julia> simulate_pixel.(ones((3,3)))
3×3 Matrix{Float64}:
 0.396078  0.96849   -0.0387851
 1.09982   0.210089   0.685605
 0.606973  1.42573   -0.675297
```

```
"""
	simulate_pixel(n_photons; read_σ=5, offset=0.5, gain=0.1)

In this function we simulate a single pixel.
It transforms a photon number to a digital unit (ADU).

The input is a integer number of photons which is then altered with poisson noise (use `pois_rand` for that).
This number is changed with additive Gauss noise (see HW01).
Finally, the result of this operation is multiplied by the linear `gain`.
At the end, we add `offset`.



* `n_photons`: number of photons
* `read_σ`: additive read noise of the sensor (applied before `gain`)
* `gain`: how much digital output per photon
* `offset`: how much digital offset

```

```julia

## Example Outputs - can vary in your case!
```julia
julia> simulate_pixel(10)
0.8570431305838304

julia> simulate_pixel(10)
2.060753446522642

julia> simulate_pixel(10)
1.020461460534897

julia> simulate_pixel(100)
8.96276245560446

julia> simulate_pixel.(ones((3,3)))
3×3 Matrix{Float64}:
 0.396078  0.96849   -0.0387851
 1.09982   0.210089   0.685605
 0.606973  1.42573   -0.675297
```
"""
function simulate_pixel(n_photons; read_σ=5, offset=0.5, gain=0.1)
    poisson_noise = pois_rand.(n_photons)
    gaussian_noise = read_σ .* randn(size(n_photons))
    noise = poisson_noise .+ gaussian_noise
    gained = noise .* gain
    signal = gained .+ offset
    return signal
```

single_output = 11.36955308442801

```julia
single_output = simulate_pixel(100)
```

123.33030000000014

```julia
mean(simulate_pixel.(ones((1000,)) * 123, read_σ=0, offset=0.0123, gain=1))
```

▸ [110.7, 135.3]

```julia
[123 - 12.3, 123 + 12.3]
```

116.06178078078086

```julia
var(simulate_pixel.(ones((1000,)) * 123, read_σ=0, offset=0.0123, gain=1))
```

▸ [110.7, 135.3]

```julia
[123 - 12.3, 123 + 12.3]
```

12.364908822646608

```julia
std(simulate_pixel.(zeros((1000,)) * 123, read_σ=12.3, offset=0, gain=1))
```

▸ [11.07, 13.53]

```julia
[12.3 - 1.23, 12.3 + 1.23]
```

535.0860767446889

```julia
std(simulate_pixel.(zeros((1000,)) * 123, read_σ=12.3, offset=0, gain=42))
```

[464.94, 568.26]

```
[516.6 - 51.66, 516.6+51.66]
```

2.1 Test

0.0123 ≈ simulate_pixel(0, read_σ = 0, offset = 0.0123, gain = 1)

```
# check offset
PlutoTest.@test 0.0123 ≈ simulate_pixel(0, read_σ=0, offset=0.0123, gain=1)
```

≈(123, mean(((x→simulate_pixel.(x; read_σ = 0, offset = 0.0123, gain = 1))).(ones((100,)) *

```
# check mean poisson
PlutoTest.@test 123 ≈ mean((x -> simulate_pixel.(x; read_σ=0, offset=0.0123,
gain=1)).(ones((100,)) * 123)) rtol=0.1
```

≈(123, var(((x→simulate_pixel.(x; read_σ = 0, offset = 0.0123, gain = 1))).(ones((100,)) * 1

```
# check variance poisson
PlutoTest.@test 123 ≈ var((x -> simulate_pixel.(x; read_σ=0, offset=0.0123, gain=1)).
(ones((100,)) * 123)) rtol=0.1
```

≈(12.3, std(((x→simulate_pixel.(x; read_σ = 0, offset = 0.0123, gain = 1))).(ones((100,)) *

```
# check readnoise
PlutoTest.@test 12.3 ≈ std((x -> simulate_pixel.(x; read_σ=0, offset=0.0123,
gain=1)).(ones((100,)) * 123)) rtol=0.1
```

≈(42 * 12.3, std(((x→simulate_pixel.(x; read_σ = 12.3, offset = 0, gain = 42))).(zeros((100(

```
# check gain
PlutoTest.@test 42 * 12.3 ≈ std((x -> simulate_pixel.(x; read_σ=12.3, offset=0,
gain=42)).(zeros((1000,))*123)) rtol=0.1
```

2.2 Variance Mean Projection

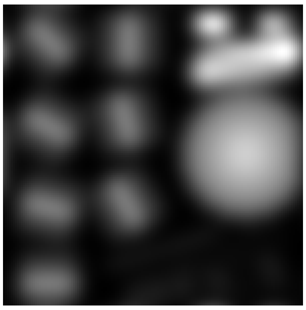
Now we want to do a Variance Mean Projection to fit the values of `offset` and `gain` since we often don't know them for real sensors.

For this test, you take a sample and in practice you defocus your microscope heavily. Via that trick, you have an image which covers all values between very dark and very bright. Hence, we expect that our image sensors both measures a large dynamic range.

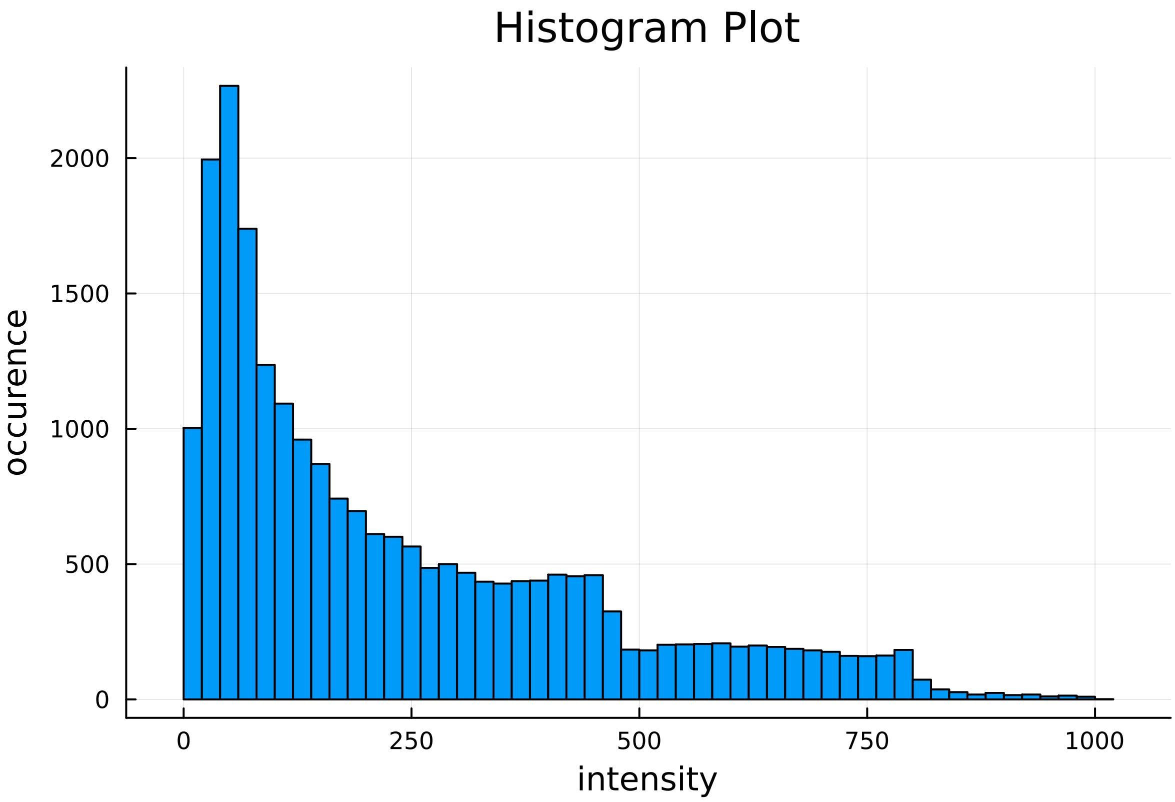
```
using FourierTools
```

```
# that's already done for you
begin
    ideal = resample(Float32.(testimage("resolution_test_512"))[50:450, 50:450],
    (150, 150)) # random amount of photons in the interval 0:500
    x_cords = range(-10, 10, length=size(ideal, 1))
    gauss = Float32.(exp.(-(x_cords.^2 .+ x_cords'.^2)))
    gauss ./= sum(gauss)
    img_blurry = conv_psf(ideal, gauss)
    img_blurry .-= minimum(img_blurry)
    img_blurry ./= maximum(img_blurry)
    img_blurry = round.(Ref(Int), img_blurry .* 1000) # round to integers
    Gray.(img_blurry ./ 500)
end;
```

That's an ideal image, where we have both very bright and very dark regions



```
# our blurry sample which emits 1000 photons in maximum
my_show(img_blurry)
```




```
histogram(img_blurry[:], xlabel="intensity", ylabel="occurence", title="Histogram
Plot", legend=nothing)
```

2.2 Task - Apply to image

Now apply the function `simulate_pixel` to `img_blurry`. Use the automatic broadcasting of Julia for that.

```
simulated_img =
150×150 Matrix{Float64}:
 2.99083  3.50343  1.89752  2.89792  …   7.65918   5.49777   3.98531  2.11414
 5.13804  3.12868  2.63481  4.21995      7.20419   4.96713   4.085    2.98867
 3.04647  3.40469  4.12338  4.32181      8.94944   7.24168   4.97596  4.86995
 4.45162  5.29623  2.37724  3.67055     12.492     8.80551   7.68381  6.81927
 5.87267  4.12868  3.50683  4.81572      9.82668  10.982     8.11079  7.0015
 5.52869  4.98652  5.29048  3.90158  …  15.2217   11.3741   10.9411   7.27603
 5.79101  6.42081  5.58361  4.27117     16.9599   13.8018    9.90212  9.64191
 ⋮                                   ⋱
 4.3552   4.24614  5.71966  6.39614      2.67628   1.80562   2.25036  2.61056
 3.97601  4.28687  4.29555  6.99228  …   2.83758   4.17655   2.79016  3.22355
 4.0145   3.8391   3.70017  5.22811      3.26172   2.60752   2.79034  3.53797
 2.82407  4.22781  4.25234  6.72661      1.98373   2.68762   2.99539  3.23128
 1.49067  2.81211  2.60929  3.58231      3.56308   3.75088   3.04687  2.70705
 3.23915  2.62102  4.48338  4.12253      6.02629   5.0067    5.11273  3.697
```

```
simulated_img = simulate_pixel.(img_blurry) # TODO, fix that line such that it uses 'simulate_pixel'. Use broadcasting mechanism
```

In practice we would need to take a few images (like 10)

Hence we repeat the image 10 times digitally

The output is an array with `size(imgs) == (150, 150, 10)`

Apply the same trick (broadcasting) for a 2D image to this 3D image

```
begin
    imgs = repeat(img_blurry, outer=(1, 1, 10));
    images = simulate_pixel.(imgs; offset, read_σ, gain);
end;
```

2.3 Task - Calculate mean and variance

Calculate the mean and variance of those images, but only along the third dimension! Either use a library (Statistics) or write it yourself :)

```
arr_test = 2×1×2 Array{Int64, 3}:
           [:, :, 1] =
            1
            2

           [:, :, 2] =
            3
            2
```

```
arr_test = [1; 2;;; 3; 2]
```

In this case, the mean along the third dimension would be

```
2×1×1 Array{Float64, 3}:
[:, :, 1] =
 2.0
 2.0
```

```
[2.0; 2.0;;;]
```

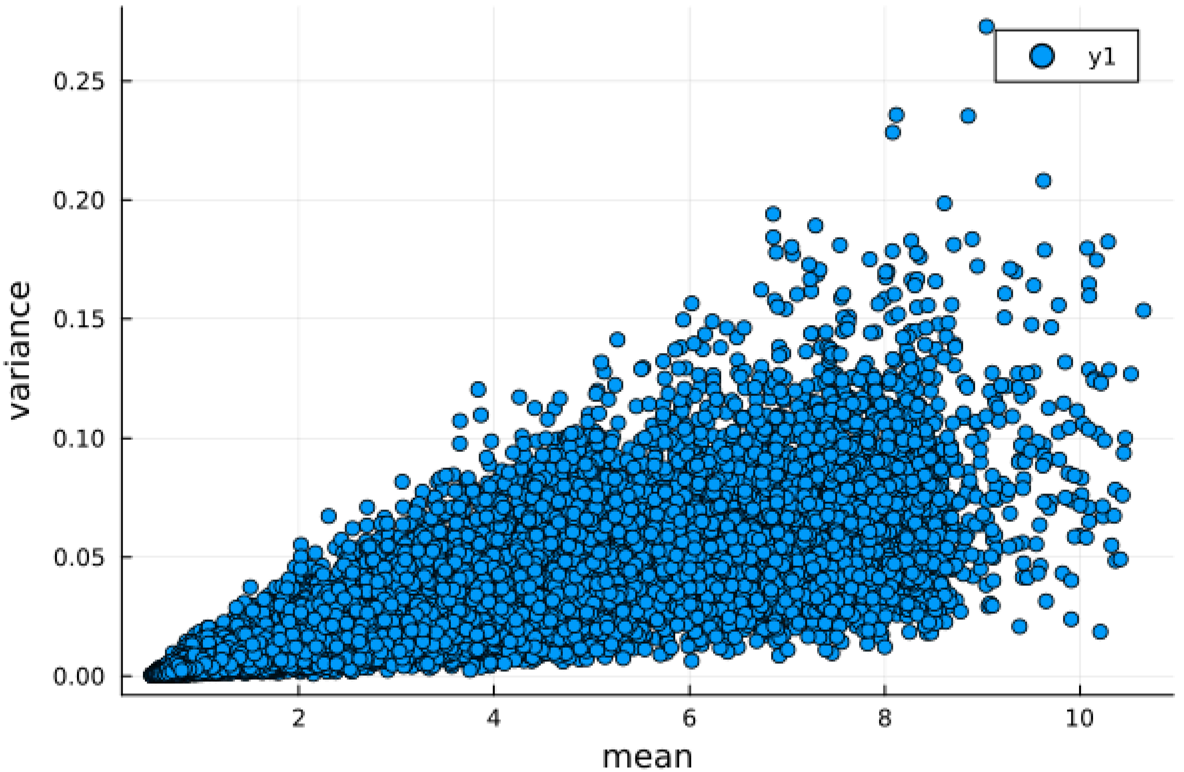
```
[0.79168, 0.793474, 0.908173, 0.882048, 1.02107, 1.05338, 1.13815, 1.29924, 1.37484, 1.5145
```

```julia
begin
    # mean
    μ_arr = mean(images, dims=3) # calculate the mean of `images` along dim 3
    μ = μ_arr[:]
end
```

```
[0.00417168, 0.0027955, 0.00638866, 0.00176376, 0.00300359, 0.00399284, 0.00677027, 0.0103
```

```julia
begin
    # variance
    variance_arr = var(images, dims=3)# calculate the variance of `images` along dim 3
    variance = variance_arr[:]
end
```

Finally, we produce a scatter plot




```julia
scatter(μ, variance, xlabel="mean", ylabel="variance")
```

2.4 Fitting

Having our noisy dataset, we want to **extract the `gain`** of the detector since the gain translate from a photon count to an eletrical signal (ADU).

Why does plotting the **variance** σ^2 over the **mean** μ allows to fit the gain? We know, that in a Poisson distribution the expected mean μ is always equal to the variance σ^2.

$$\sigma^2 = \mu$$

If we now take 10 captures of the same image, we always expect the same mean μ for a certain pixel. Of course, those are 10 evaluation of a random measurement process, therefore we observe fluctuations. Hence, for each pixel we can calculate the mean and the variance.

Since the detector multiplies the measured photons with a certain gain, we do not measure the photons but the digital value. Hence the **measured mean** is

$$\hat{\mu} = \text{gain} \cdot \mu = \text{gain} \cdot \sigma^2$$

where μ would be the **expected photon number**.

However, the **measured variance** is

$$\hat{\sigma}^2 = \sum_i (\hat{\mu} - \text{gain} \cdot x_i)^2 = \text{gain}^2 \sigma^2$$

where σ^2 would be the variance of the photon number.

$$\hat{\sigma}^2 = \text{gain} \cdot \hat{\mu} + \text{offset}$$

The slope of our variance over mean is then

$$\frac{\Delta \hat{\sigma}^2}{\Delta \hat{\mu}} = \text{gain}$$

Dividing $\hat{\mu}$ by **gain** returns the measured photon number.

```
md" ## 2.4 Fitting
Having our noisy dataset, we want to **extract the `gain`** of the detector since the gain translate from a photon count to an eletrical signal (ADU).

Why does plotting the **variance** $\sigma^2$ over the **mean** $\mu$ allows to fit the gain?
We know, that in a Poisson distribution the expected mean $\mu$ is always equal to the variance $\sigma^2$.

$\sigma^2 = \mu$

If we now take 10 captures of the same image, we always expect the same mean $\mu$ for a certain pixel. Of course, those are 10 evaluation of a random measurement process, therefore we observe fluctuations. Hence, for each pixel we can calculate the mean and the variance.


```

```
Since the detector multiplies the measured photons with a certain gain, we do not
measure the photons but the digital value.
Hence the **measured mean** is

$\hat \mu = \text{gain} \cdot \mu = \text{gain} \cdot \sigma^2$

where $\mu$ would be the **expected photon number**.

However, the **measured variance** is

$$\hat \sigma^2 = \sum_{i} (\hat \mu - \text{gain} \cdot x_i)^2 = \text{gain}^2
\sigma^2$$

where $\sigma^2$ would be the variance of the photon number.

$\hat\sigma^2 = \text{gain} \cdot \hat\mu + \text{offset}$

The slope of our variance over mean is then

$$\frac{\Delta \hat \sigma^2}{\Delta \hat \mu} = \text{gain}$$

Dividing $\hat \mu$ by $\text{gain}$ returns the measured photon number.
```

```
# for fitting
using LsqFit
```

2.5 LsqFit.jl

We now want to fit a linear function with a slope and a offset on our data. Try to find out here how that works

```
model (generic function with 1 method)
```

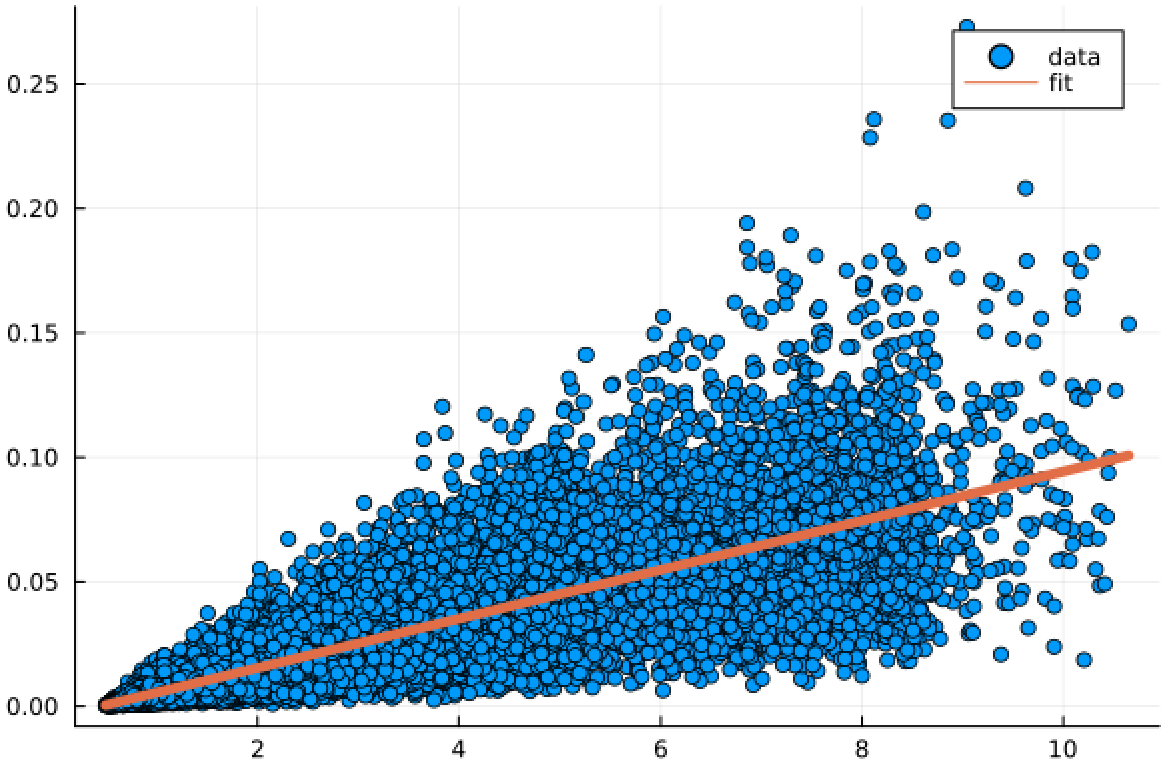
```
@. model(x, p) = p[1] .* x .+ p[2] #p[1] is gain, p[2] offset
```

```
p0 = [0.5, 0.5]
```

```
p0 = [0.5, 0.5] # TODO (maybe you need to convert it to Float32, maybe not), initial guess
```

```
fit =
LsqFitResult([0.00987107, -0.00439468], [-0.000751632, 0.000642253, -0.0018187, 0.00254832,
```

```
fit = curve_fit(model, μ, variance, p0) # TODO
```




```
begin
    scatter(μ, variance, label="data")
    plot!(μ, model(μ, fit.param), linewidth=5, label="fit")
end
```

Final Check

If you slide those parameters, you change the simulated values.

offset = 0.5

read_σ = 2

gain = 0.01

The final solution is gain = 0.009871 ± 4.7e-5

```
md"The final solution is gain = $(round(fit.param[1], sigdigits=4)) ±
$(round(stderror(fit)[1], sigdigits=2))"
```